

STREAMING FETCH

Cursor-free scans that let the remote server choose a parallel plan

RAFIA SABIH

SR. POSTGRESQL DEVELOPER



DATABASE PRODUCTS & TOOLS































AGENDA

1. CURRENT POSTGRES_FDW SCAN MODEL
   What are its pros and cons

2. THE NEW FETCH MECHANISM
   Implementation details

3. LIMITATIONS



CURRENT POSTGRES_FDW MODEL

CURSORS

- Every foreign scan is a cursor

```
1. -- BeginForeignScan / first Iterate
2. DECLARE c1 CURSOR FOR
3.   SELECT a, b FROM public.t1 WHERE ...;
4.
5. -- each time the batch runs out
6. FETCH 100 FROM c1;
7. FETCH 100 FROM c1;
8. ...
9.
10. -- EndForeignScan
```



CURRENT POSTGRES_FDW MODEL

CURSORS: PROS

- Cursors give control
- Multiple cursors can be open at a time
- Only fetch_size number of tuples are fetched at a time

- It works, reliably!



CURRENT POSTGRES_FDW MODEL

CURSORS: CONS

- When declared as cursor, parallel plans are prohibited
- Query planning happens without CURSOR_OPT_PARALLEL_OK
- Query with cursor fetches in small batches
- Parallel query cannot pause and resume like that



NEW FETCH MECHANISM

STREAMING_FETCH

- Can be declared at server or table level
- Default is OFF

```
ALTER SERVER remote
  OPTIONS (ADD streaming_fetch 'true');

-- a table setting overrides the server
ALTER FOREIGN TABLE small_lookup
  OPTIONS (ADD streaming_fetch 'false');

EXPLAIN (VERBOSE) SELECT ...;
  Remote SQL: SELECT ...
  Streaming Fetch: true

```



NEW FETCH MECHANISM

STREAMING_FETCH

- No Cursors, No Declare, No Fetch, No Close
- Query starts and runs till completion
- Scan can still get fetch_size number of tuples at a time as before
- Parallel plans are possible



NEW FETCH MECHANISM

STREAMING_FETCH

- Every connection knows its active_scan
- It is set in init_scan
- Cleared once the query is completed

```
1. typedef struct PgFdwConnState
2. {
3.   AsyncRequest   *pendingAreq;
4.   PgFdwScanState *active_scan;
5. } PgFdwConnState;
6.
```



NEW FETCH MECHANISM

STREAMING_FETCH

- Read every remaining tuples of the active_scan
- Tuples are stored in tuplestore
- At the next read, fetch_more_data fetches tuples from the tuplestore

```
1. typedef struct PgFdwScanState
2. …
3. /* Fields required for the streaming_fetch mode only */
4. bool   streaming_fetch;
5. Tuplestorestate *tuplestore;
6. …
7. } PgFdwScanState;
```



NEW FETCH MECHANISM

STREAMING_FETCH

- Cursors share connection
- More than one cursor can be open at the same time
- Stream query can run only one at a time



NEW FETCH MECHANISM

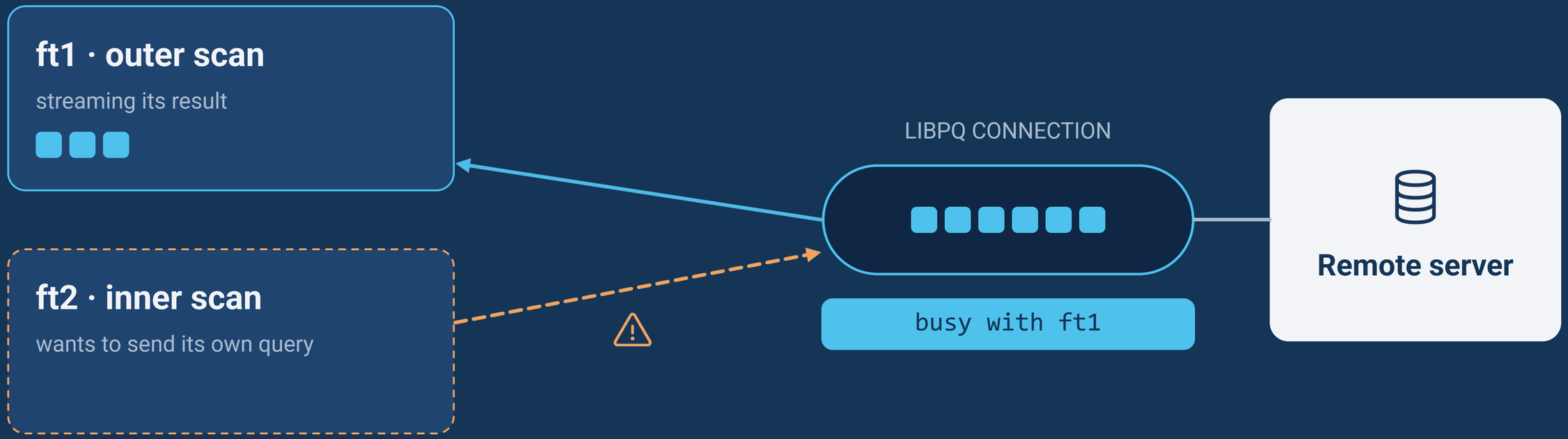
STREAMING_FETCH

```
1. Nested Loop
2.   -> Foreign Scan on ft1 (streaming)
3.   -> Foreign Scan on ft2
4.        Filter: ft2.x = ft1.x
5.
6. -- same server, same user
7. -- = same connection
```



NEW FETCH MECHANISM

ONE CONNECTION, TWO SCANS



- Same foreign server + same user mapping = one shared libpq connection.
- A connection carries one result stream at a time, so ft1 must hand it over without losing its rows.

NEW FETCH MECHANISM

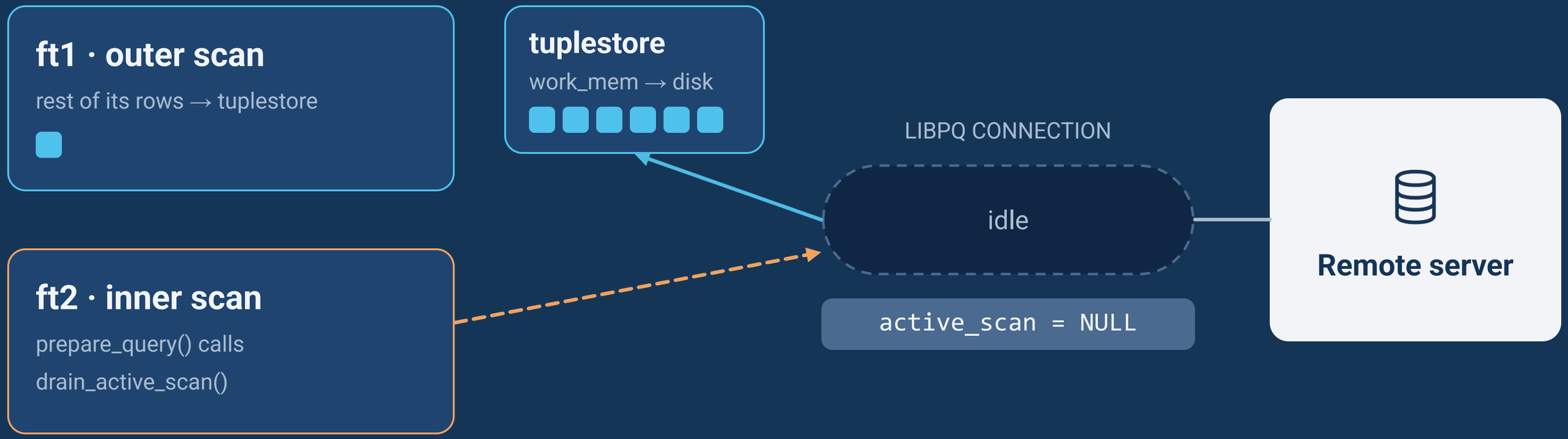
STEP 1 · FT1 IS STREAMING



- init_scan() sends ft1’s query; the remote server starts streaming rows.
- ft1 has read chunk 1. The rest of its rows are still in flight on the connection.

NEW FETCH MECHANISM

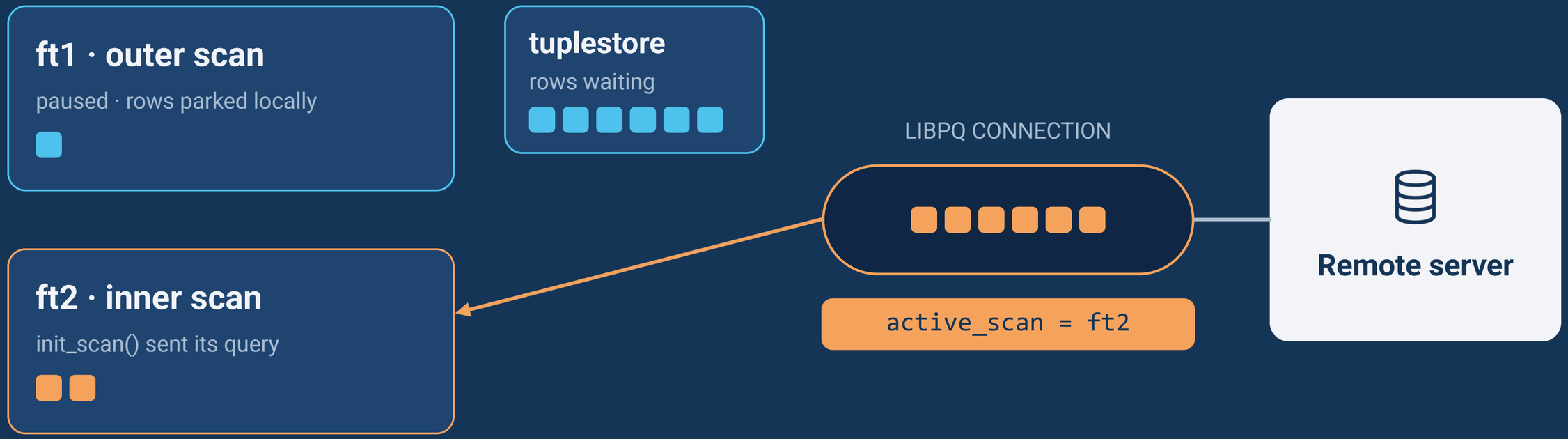
STEP 2 · FT2 NEEDS THE CONNECTION



- Before ft2 sends anything, its prepare_query() calls drain_active_scan().
- ft1’s remaining rows are read to the end and copied into ft1’s tuplestore (work_mem, then disk).

NEW FETCH MECHANISM

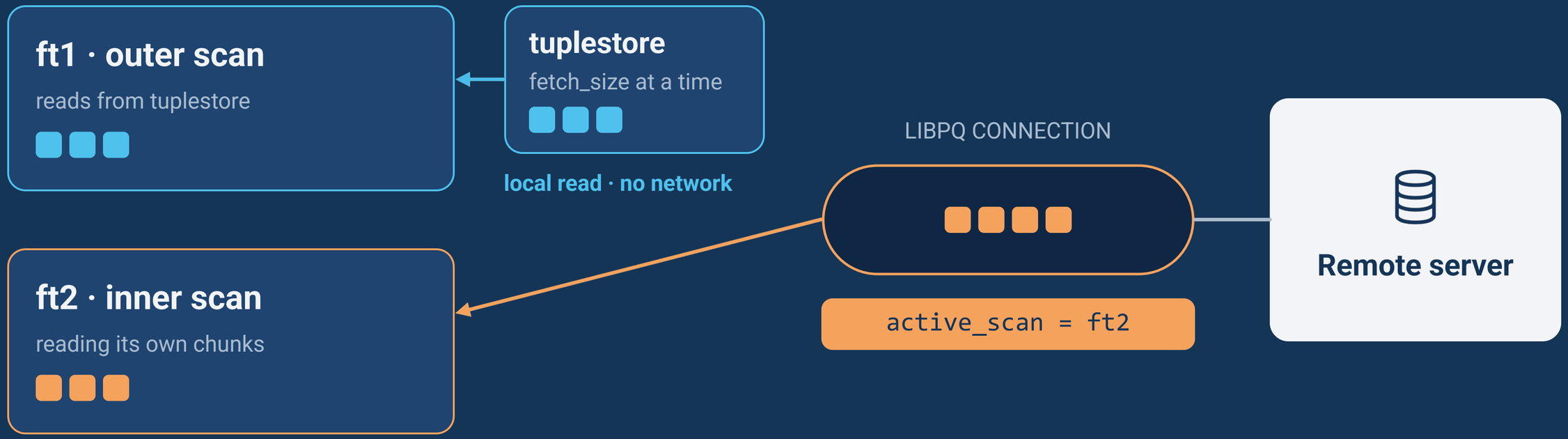
STEP 3 · FT2 RUNS ITS QUERY



- ft2’s init_scan() sends its query and reads its own chunks; active_scan = ft2.
- ft1’s rows wait in its tuplestore: nothing is lost, nothing is fetched twice.

NEW FETCH MECHANISM

STEP 4 · FT1 ASKS FOR MORE ROWS



- fetch_more_data() gives ft1 its next fetch_size rows from the tuplestore, with no network round trip.
- Before ft2’s next rescan, it first discards or drains its own leftover stream.

NEW FETCH MECHANISM

MULTIPLE SCANS: AT A GLANCE

1

ft1 streams

init_scan() sends the query; ft1 reads chunk 1.

```
active_scan = ft1
```

2

Drain ft1

ft2 calls drain_active_scan(); ft1's rows move into its tuplestore.

```
active_scan = NULL
```

3

ft2 takes over

ft2 sends its query; the connection now carries only ft2's rows.

```
active_scan = ft2
```

4

ft1 reads locally

fetch_more_data() takes fetch_size rows from the tuplestore: no round trip.

```
active_scan = ft2
```

ft1's rows are never lost and never fetched twice: they wait in its tuplestore until ft1 asks for them.



NEW FETCH MECHANISM

RESCAN: THE SCENARIO

```
Nested Loop
  -> Foreign Scan on ft1
  -> Foreign Scan on ft2
       Filter: x = $1
```

$1 = the x value of the current ft1 row

ft1 row: x = 1 → First scan: send ft2's query with $1 = 1

ft1 row: x = 2 → Rescan, new $1: ft2 needs different rows

ft1 row: x = 2 → Rescan, same $1: ft2 needs the same rows again

Each restart calls ReScanForeignScan() on ft2. The question is: can ft2 reuse what it has, or must it ask the remote again?



NEW FETCH MECHANISM

RESCAN: WITH STREAMING_FETCH

1	Did the parameters change?	**Yes: discard, re-send** new remote query
2	Same parameters, and every row came in the first batch?	**Yes: rewind in memory** no network at all
3	Anything else: several batches read, or rows went through the tuplestore	**Discard, re-send** new remote query

“Discard” = drop the batch in memory and, if ft2’s old stream is still on the connection, read its leftover rows and throw them away. Only then can the query be sent again.



NEW FETCH MECHANISM

RESCAN: CURSORS VS STREAMING

Situation	Cursor mode	streaming_fetch
Parameters changed	CLOSE, re-DECLARE	discard, re-send
Read more than one batch	MOVE BACKWARD ALL (< v15) or re-DECLARE	
Everything in one batch	rewind in memory	rewind in memory
Rows came via the tuplestore	can’t happen	discard, re-send

Same in most cases: both modes ask the remote to run the query again, or both rewind in memory.

Extra cost for streaming: an unfinished old stream must be read to the end before the new query goes out.



NEW FETCH MECHANISM

STREAMING_FETCH: PERFORMANCE

- max_parallel_workers_per_gather = 4

  shared_buffers = 2GB

  work_mem = 64MB
- Two foreign servers identical except streaming_fetch; use_remote_estimate on
- big: 10M rows (int, int, numeric, md5 text, timestamptz) ·

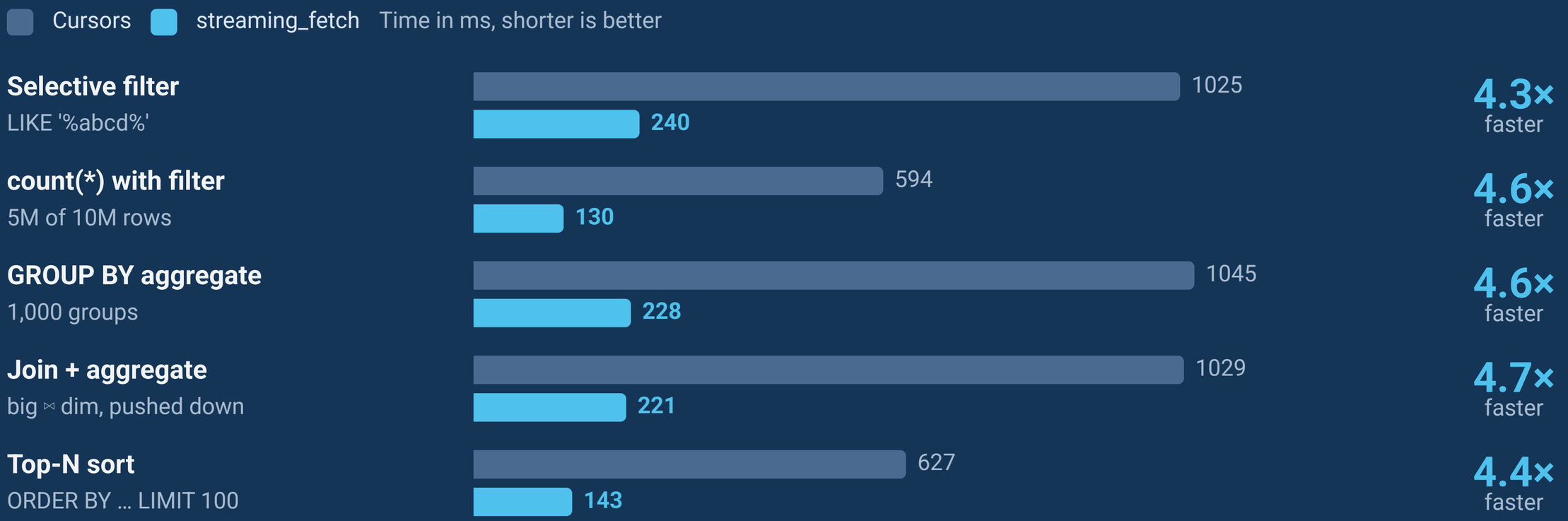
  dim: 1k rows

  small: 100k rows



NEW FETCH MECHANISM

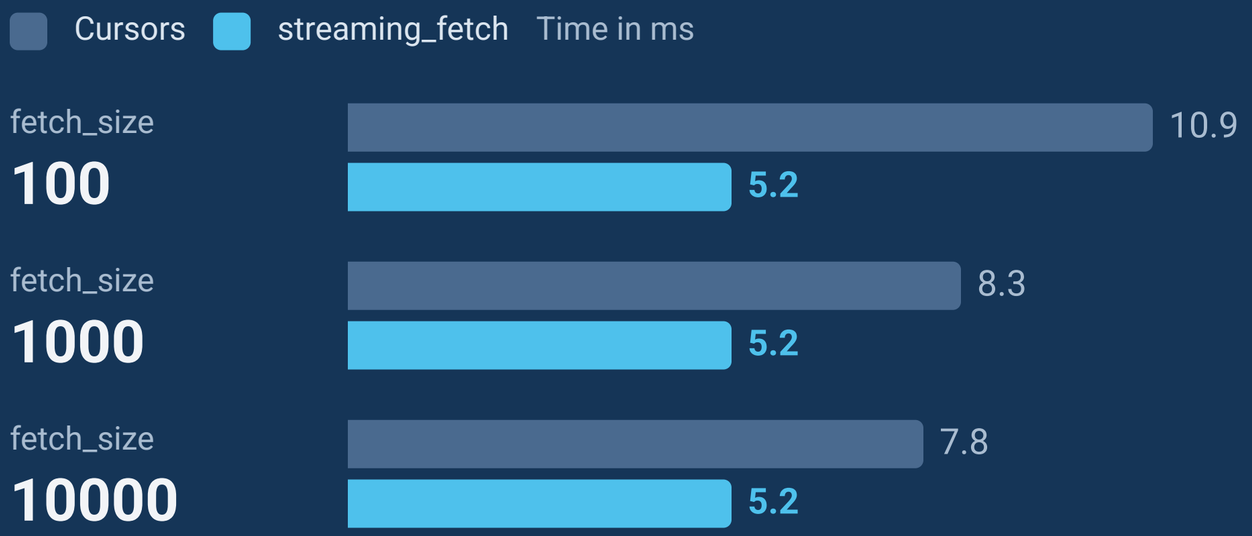
PERFORMANCE: PARALLEL QUERY



Setup: 4 parallel workers per gather · work_mem 64MB · shared_buffers 2GB · use_remote_estimate on · big = 10M rows

NEW FETCH MECHANISM

PERFORMANCE: NO ROUND TRIPS



5.2 ms
at every fetch_size: one query, the rows just keep coming

Cursors pay a round trip per FETCH: smaller batches, more trips

Faster even without parallel query, for large scans

NEW FETCH MECHANISM

PERFORMANCE: WHEN IT REGRESSES

SELECT * FROM big WHERE random() < 2 LIMIT 10

streaming_fetch reads and discards 10M rows before freeing the connection

LIMIT can not be pushed down

Cursor: 0.3 ms
streaming_fetch: 3148 ms



NEW FETCH MECHANISM

STREAMING_FETCH: TAKEAWAY

- Beneficial for large scans
- Performant when filters, aggregates, joins pushed to remote server
- Better when FETCH round trips might be expensive

- Cursors are better when queries stop early



NEW FETCH MECHANISM

STREAMING_FETCH: LIMITATIONS

- Async is not possible
- A parametrized inner scan sends the full query again
- Tuplestore may spill to disk
- LIMIT that is not pushed down still reads whole result



NEW FETCH MECHANISM

STREAMING_FETCH: NEXT

- Patch is active on the hackers list
- Registered in the current Commitfest
- Reviews are welcome!

- People involved: Bernd Helmle, Robert Haas
  Kenan Yelmaz, Alexander Pyhalov, Yilin Zhang



Questions?

Link for feedback



Keep Streaming!

Thank You!

